Title

A Hartree-Fock Application Using UPC++ and the New DArray Library

Permalink

https://escholarship.org/uc/item/1pz877d3

ISBN

9781509021406

Authors

Ozog, David
Kamil, Amir
Zheng, Yili
et al.

Publication Date

2016-05-01

DOI

10.1109/ipdps.2016.108

Peer reviewed

A Hartree-Fock Application using UPC++ and the New DArray Library

David Ozog*, Amir Kamil†, Yili Zheng†, Paul Hargrove†, Jeff R. Hammond‡, Allen Malony*, Wibe de Jong†, Kathy Yelick†

**University of Oregon, Eugene, Oregon USA*
‡Intel Corporation, Portland, Oregon USA
†Lawrence Berkeley National Laboratory, Berkeley, California USA

ozog@uoregon.edu, akamil@lbl.gov, yzheng@lbl.gov, phhargrove@lbl.gov, jeff.r.hammond@intel.com, malony@uoregon.edu, wadejong@lbl.gov, kayelick@lbl.gov

***Abstract*—The Hartree-Fock (HF) method is the fundamental first step for incorporating quantum mechanics into many-electron simulations of atoms and molecules, and it is an important component of computational chemistry toolkits like NWChem. The GTFock code is an HF implementation that, while it does not have all the features in NWChem, represents crucial algorithmic advances that reduce communication and improve load balance by doing an up-front static partitioning of tasks, followed by work stealing whenever necessary.**

To enable innovations in algorithms and exploit next generation exascale systems, it is crucial to support quantum chemistry codes using expressive and convenient programming models and runtime systems that are also efficient and scalable. This paper presents an HF implementation similar to GTFock using UPC++, a partitioned global address space model that includes flexible communication, asynchronous remote computation, and a powerful multidimensional array library. UPC++ offers runtime features that are useful for HF such as active messages, a rich calculus for array operations, hardware-supported fetch-and-add, and functions for ensuring asynchronous runtime progress. We present a new distributed array abstraction, DArray, that is convenient for the kinds of random-access array updates and linear algebra operations on block-distributed arrays with irregular data ownership. We analyze the performance of atomic fetch-and-add operations (relevant for load balancing) and runtime attentiveness, then compare various techniques and optimizations for each. Our optimized implementation of HF using UPC++ and the DArrays library shows up to 20% improvement over GTFock with Global Arrays at scales up to 24,000 cores.

***Keywords*-Hartree-Fock, self-consistent field (SCF), quantum chemistry, PGAS, UPC/UPC++, Global Arrays, performance analysis, load balancing, work stealing, attentiveness**

I. Introduction

In order to develop the next generation of materials and chemical technologies, molecular simulations must incorporate quantum mechanics to effectively predict useful properties and phenomena. However, chemical simulations that include quantum effects are computationally expensive and frequently scale superlinearly in the number of atoms in the system simulated. Even for a relatively modest system like the graphene unit in Figure 1, substantial computing power is required for an accurate treatment of quantum effects. Despite the opportunities for exploiting parallelism, even the most sophisticated and mature chemistry software tools exhibit scalability problems due to the inherent load imbalance and difficulty in exploiting data locality in quantum chemistry methods.

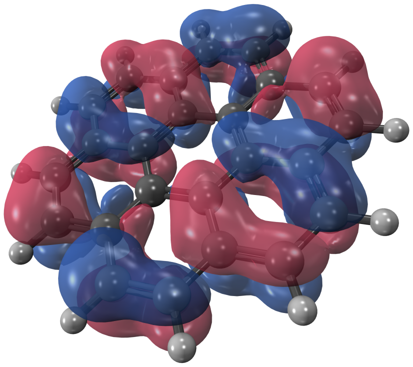


Figure 1: A graphene subunit with 36 atoms. The blue and red blobs show the lowest unoccupied molecular orbital based on a Hartree-Fock calculation.

The Hartree-Fock (HF) method is the quintessential starting point for doing such *ab initio* calculations, so most all quantum chemistry codes provide some sort of HF implementation. Additionally, the more accurate and very popular *post-Hartree-Fock* methods, such as coupled cluster and many-body perturbation theory, depend on the reference wavefunction provided by HF. Nearly all quantum chemistry codes base their post-Hartree-Fock software implementations on the programming model and data structures used in the corresponding HF component. It is therefore vital, when developing modern quantum chemistry codes, to start from the ground-up with very efficient and scalable programming constructs in the HF modules.

Unfortunately, the most well-used and scalable HF and post-HF codes often utilize programming models that do not embrace modern software capabilities, such as those provided by C++11 and C++14. For example, NWChem is generally considered to be the most scalable of all quantum chemistry codes [1],[2], yet it uses a toolkit called Global

Arrays (explained fully in section II-B), which only supports distributed arrays containing elements of type `int`, `float`, `double`, or `complex`. Even new codes such as GTFock, which introduce impressive algorithmic enhancements to HF, still use the same Global Arrays programming model without support for features like distributed structures of objects, active messages, and remote memory management. UPC++ is parallel programming library that supports these modern features. Others have recently utilized these capabilities of UPC++ for scientific calculations, while achieving comparable or better performance than similar implementations that use only MPI [3], [4].

This paper explores the use of UPC++, a partitioned global address space (PGAS) extension for C++, for doing HF calculations. We base our implementation on the GTFock code, but instead of using the Global Arrays (GA) library for operating on distributed data structures, we use a new DArray library written in UPC++. The DArray library incorporates the functionality of GA while supplanting it with new features, such as the ability to apply user-defined functions across distributed tiles of the global data structure. Using DArrays, our UPC++ HF application accomplishes the same algorithmic improvements as GTFock, while making efficient use of unique DArray capabilities. Our measurements show that the UPC++ version of Hartree-Fock achieves as much as 20% performance improvement over GTFock with up to 24,000 processor cores.

II. Background

A. The Hartree-Fock Method

The goal of the Hartree-Fock (HF) method is to approximately solve the many-body time-independent Schrödinger equation, $\mathbf{H}\left|\psi\right\rangle = E\left|\psi\right\rangle$, where $\mathbf{H}$ is the Hamiltonian operator, which extracts the sum of all kinetic and potential energies, E, from the wavefunction, $\left|\psi\right\rangle$. Here we make the standard set of assumptions: the Born-Oppenheimer approximation (in which nuclei are fixed but electrons move freely), Slater determinant wavefunctions (that easily satisfy the anti-symmetry principle), and non-relativistic conditions. After these approximations, our focus resides only on the electronic terms of the Hamiltonian and the wavefunction: $\mathbf{H}_{elec}\left|\psi_{elec}\right\rangle = E\left|\psi_{elec}\right\rangle$.

In both HF and post-HF methods, the molecular orbitals that express the electronic wavefunction consist of a sum of *primitive basis functions* from set $\{\phi_j\}$. We desire $\{\phi_j\}$ to be complete, but this is not practical since it generally requires an infinite number of functions. We therefore truncate to a finite n value large enough to balance the trade-off between accuracy and computational cost:

$$\left|\psi_i\right\rangle = \sum_{j=1}^{n} c_{ij}\left|\phi_j\right\rangle$$

Typically, each primitive is a Gaussian function centered at the nucleus location. The HF method attempts to determine the c_{ij} values that best minimize the ground state energy in accordance with the variational principle. By utilizing a numerical technique for iteratively converging the energy, each subsequent iteration becomes more and more consistent with the field that is imposed by the input molecule and basis set. The method is accordingly called the self-consistent field (SCF) method, and Algorithm 1 shows its de facto procedure in pseudocode.

Algorithm 1 The SCF Procedure

```
Inputs:
    1) A molecule (nuclear coordinates, atomic numbers, and $N$ electrons)
    2) A set of basis functions $\{\phi_\mu\}$
Outputs:
    Final energy $E$, Fock matrix $\mathbf{F}$, density matrix $\mathbf{D}$, coefficient matrix $\mathbf{C}$

 1: Calculate overlap integrals $S_{\mu\nu}$, core Hamiltonian terms $H^{core}_{\mu\nu}$, and
    the two-electron integrals $(\mu\nu|\lambda\sigma)$.
 2: Diagonalize the overlap matrix $\mathbf{S} = \mathbf{UsU}^\dagger$ and obtain $\mathbf{X} = \mathbf{Us}^{1/2}$.
 3: Guess the initial density matrix $\mathbf{D}$.
 4: while $E$ not yet converged do
 5:     Calculate $\mathbf{F}$ from $H_{\mu\nu}$, $\mathbf{D}$, and $(\mu\nu|\lambda\sigma)$.
 6:     Transform $\mathbf{F}$ via $\mathbf{F}' = \mathbf{X}^\dagger\mathbf{F}\mathbf{X}$.
 7:     $E = \sum_{\mu,\nu} D_{\mu\nu}(H^{core}_{\mu\nu} + F_{\mu\nu})$
 8:     Diagonalize $\mathbf{F}' = \mathbf{C}'\epsilon\mathbf{C}'^\dagger$.
 9:     $\mathbf{C} = \mathbf{XC}'$
10:     Form $\mathbf{D}$ from $\mathbf{C}$ by $D_{\mu\nu} = 2\sum_i^{N/2} C_{\mu i}C^*_{\nu i}$.
11: end while
```

Many SCF iterations are required for energy convergence, so steps 1-3 of Algorithm 1 cost much less than steps 5-10. Step 5 normally comprises a majority of the execution time in Hartree-Fock codes, because each element of the Fock matrix requires computing several two-electron integrals:

$$F_{ij} = H^{core}_{ij} + \sum_{\lambda\sigma}(2(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma))$$

The two-electron integrals on the right-hand side are plentiful and expensive to compute [5]. They take this form:

$$(\mu\nu|\lambda\sigma) = \iint \phi^*_\mu(\mathbf{r}_1)\phi_\nu(\mathbf{r}_1)r^{-1}_{12}\phi^*_\lambda(\mathbf{r}_2)\phi_\sigma(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2 \quad (1)$$

As mentioned, there are formally $\mathcal{O}(n^4)$ integrals to compute within the basis set. However, from the definition in 1, we see there are only $\sim n^4/8$ unique integrals by permutational symmetry and the number of non-zero integrals is asymptotically $\mathcal{O}(n^2)$ when Schwartz screening is employed.

While the two-electron integrals comprise the most computation time in SCF, communication and synchronization overheads can very well dominate, particularly at large scale. As discussed below (Section II-E), inherent communication costs arise from the parallel decomposition of HF codes, leading to load imbalance and synchronization delays. Parallel HF applications also exhibit highly diverse accesses across distributed process memory spaces. As such, HF is well-suited for a programming model that emphasizes lightweight and one-sided communication within a single global address space. This is the subject of the next section.

B. PGAS in Quantum Chemistry

The algorithmic characteristics and resource requirements of HF (and post-HF) methods clearly motivate the use of distributed computation. HF tasks are independent and free to be computed by any available processor. Also, simulating a molecule of moderate size has considerable memory requirements that can easily exceed the memory space of a single compute node. However, at a high level of abstraction, *indexing into distributed HF data structures need not be different than indexing shared memory structures*. This programming abstraction is the basis of the partitioned global address space (PGAS) model for distributing and interacting with data. In computational chemistry, this model is advantageous for interacting with arrays and tensors productively and efficiently.

The PGAS model utilizes one-sided communication semantics, allowing a process to access remote data with a single communication routine. Remote memory access (RMA) is particularly advantageous in HF applications for three primary reasons. First, HF exhibits highly irregular access patterns due to the intrinsic structure of molecules and the necessary removal of integrals from the Fock matrix in a procedure called "screening." Second, there is a need for efficient atomic accumulate operations to update tiles of the global matrices without the explicit participation of the target process. Finally, dynamic load balancing in HF is usually controlled by either a single atomic counter [1], or many counters [6], [7], both of which require a fast one-sided fetch-and-add implementation.

The NWChem software toolkit [8] paved the way towards the ubiquity of PGAS in computational chemistry using the Global Arrays (GA) library and the underlying ARMCI messaging infrastructure [1], [2]. GTFock followed suit, also using GA for tiled accesses across the irregularly distributed Fock and density matrices. The GA model is applicable to many applications, including ghost cells and distributed linear algebra, but GA's primary use today is for quantum chemistry. GA is limited to a C and Fortran API, but does have Python and C++ wrapper interfaces. Besides UPC++, which is the subject of the next section, there exist many other PGAS runtimes, including but not limited to: Titanium, X10, Chapel, Co-Array Fortran, and UPC.

C. UPC++ and Multidimensional Arrays

UPC++ is a C++ library for parallel programming with the PGAS model. It includes features such as global memory management, one-sided communication, remote function invocation and multidimensional arrays [3].

1) Multidimensional Arrays: A general multidimensional array abstraction is very important for scientific applications, but unfortunately, the support for multidimensional arrays in the C++ standard library is limited [9]. The UPC++ library implements a multidimensional array abstraction that incorporates many features that are useful in the PGAS setting, and we build our own distributed array representation on top of UPC++ multidimensional arrays. Here, we briefly introduce the concepts that are relevant to implementing the distributed arrays used in the HF algorithm:

- A *point* is a set of N integer coordinates, representing a location in N-dimensional space. The following is an example of a *point* in three-dimensional space:

  ```
  point<3> p = {{ -1, 3, 2 }};
  ```

- A *rectangular domain* (or *rectdomain*) is a regular set of points between a given lower bound and upper bound, with an optional stride in each dimension. Rectangular domains represent index sets, and the following is an example of the set of points between $(1, 1)$, inclusive, and $(4, 4)$, exclusive:

  ```
  rdomain<2> rd( PT(1,1), PT(4,4) );
  ```

- A UPC++ multidimensional array, represented by the `ndarray` class template, is a mapping of points in a rectdomain to elements. The following creates a two-dimensional `double` array over the rectdomain above:

  ```
  ndarray<double,2> A( rd );
  ```

UPC++ supports a very powerful set of operations over domains, including union, intersection, translation, and permutation. Since UPC++ multidimensional arrays can be created over any rectangular domain, these domain operations simplify the expression of many common array operations. New views of an array's data can also be created with transformations on its domain. For example, the following code transposes an array `A` into a new array `B` by creating `B` over a transpose of `A`'s domain, creating a transposed view of `A`, and copying that view into `B`:

```
rectdomain<ndim> permuted_rd =
    A.domain().transpose();
ndarray<T, ndim> B( permuted_rd );
B.copy(A.transpose());
```

An `ndarray` represents an array that is stored in a single memory space. However, the memory space containing an `ndarray` may be owned by a remote process; the `global` template parameter is used to indicate that a given `ndarray` may reside in a remote memory space:

```
ndarray<T, ndim, global> remote_array;
```

In keeping with the PGAS model, UPC++ supports one-sided remote access to an `ndarray`, both at the granularity of individual elements and, using the `copy` operation, a set of multiple elements.

The one-sided `copy` operation on `ndarrays` is an especially powerful feature. In the call `B.copy(A)`, the library automatically computes the intersection of the domains of `A` and `B`, packs the elements of `A` that are in that intersection if necessary, transfers the data from the process that owns `A` to the process that owns `B`, and unpacks the elements into the appropriate locations in `B`. Active messages are used to make this process appear seamless to the user, and the library also supports non-blocking transfers using the `async_copy()` method on `ndarrays`.

A final domain and array feature used in the HF code is the `upcxx_foreach` construct, which iterates over the points in a domain. This allows the expression of dimensionality-independent loops, such as the following code that sets all elements of an array `A` to 0.0:[1]

```
upcxx_foreach( pt, A ) { A[pt] = 0.0; };
```

2) Other UPC++ Features: UPC++ shared arrays, a feature inherited from the Unified Parallel C (UPC) language, are simple 1D arrays block-cyclically distributed across all processes. The following declares a shared array:

```
shared_array<T> A;
```

where `T` is the type of the array elements and should be *trivially copyable*. Before its first use, a shared array must be explicitly initialized by calling `init()` with the array size and block size as arguments. The creation of a shared array is collective, which means that the set of processes storing the array must agree on the same array size and block size. Shared arrays are limited in that they are only one-dimensional and have fixed block sizes.

To help move computation and save communication, UPC++ extends C++11's `async` feature for distributed-memory systems, enabling functions to execute on any node in a cluster. UPC++ uses C++11 variadic templates to package function arguments together with the function object and ships the closure to a remote node via GASNet [10].

D. Distributed Arrays with Irregular Ownership

HF and post-HF methods require storing large, irregularly distributed arrays. Figure 2 shows examples of *regularly* distributed arrays with round-robin and 2D block-cyclic assignments, and *irregularly* distributed arrays with 2D blocks. In HF, the irregular distributions arise after screening out negligible integral quantities. For example, the top right of Figure 2 may represent the initial assignment of the Fock matrix to processes. However, after determining the number of non-zeros in the collection of all shell pairs, the matrix is *repartitioned* so that each process is assigned approximately the same number of non-zero tasks. This new distribution might look more like the bottom left of Figure 2.

Like GA, UPC++ shared arrays can be either regularly or irregularly distributed, as shown in Figure 2. Unlike GA, element assignment in UPC++ is round robin by default; however, irregular arrays with `N` dimensions containing elements of type `T` can easily be created in this manner:

```
shared_array< ndarray<T, N> > A;
A.init( my_local_array );
```

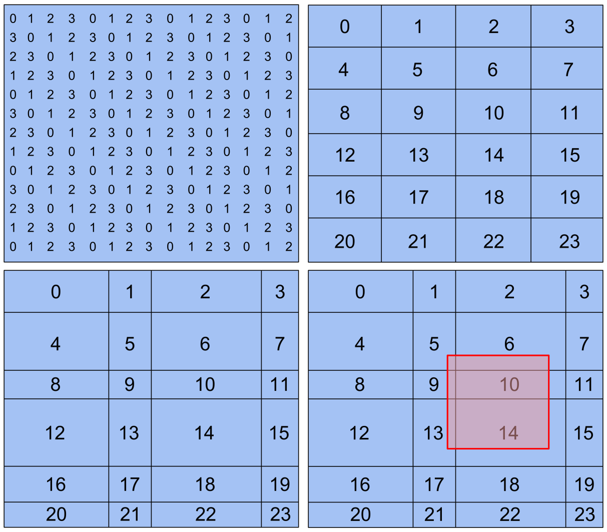


Figure 2: By default, UPC++ uses a regular round-robin distribution (upper left), while GA uses a regular 2D block-cyclic distribution (upper right). After screening and repartitioning, HF arrays are *irregularly* distributed (bottom left). A tiled access (in red) may span multiple owning processes (bottom right), in this case ranks 5, 6, 9, 10, 13, and 14.

where each rank defines `my_local_array` across its relevant portion of the global rectdomain. GA creates a similar structure using the `NGA_Create_irreg` API call. GA supports arbitrary tiled accesses from the global array; however, prior to this work, UPC++ did not support such accesses unless done for each owning process *explicitly*. More specifically, if `tile` is the object represented in red in Figure 2, then a `get` call would need to occur for *all* ranks 5, 6, 9, 10, 13, and 14. This motivates a higher level of abstraction when doing tiled access operations. As a result, we implemented a *distributed* version of the multidimensional array library, the implementation of which is the subject of Section III-B.

E. Load Balancing in Quantum Chemistry

The primary hindrance to scalability in quantum chemistry codes is often due to load imbalance [11], [12], [7]. Although individual two-electron integrals do not possess drastic differences in execution time, bundles of *shell quartets* can vary greatly in cost. It is necessary to designate shell quartets as task units in HF codes because it enables the reuse of intermediate quantities shared by basis functions within a quartet [13], [6]. The goal is to assign these task units to processors with minimal overhead and a schedule that reduces the makespan.

NWChem balances load with a centralized dynamic scheduler, which is controlled by a single global counter control referred to as `nxtval` (for "next value"). Each task has a unique ID, and a call to `nxtval` *fetches* the current

[1] This example is just for illustration, since the call `A.set(0.0)` accomplishes this much more concisely.

ID of an uncomputed task, then atomically *adds* 1 to the counter so the next task gets executed. Once the counter reaches the total number of tasks, all work is done. For this reason, the performance of RMA fetch-and-add operations is very important for the scalability of computational chemistry codes like NWChem and GTFock. This has motivated the implementation and analysis of hardware-supported fetch-and-ops on modern interconnects, such as Cray Aries using the GNI/DMAPP interfaces [14].

The `nxtval` scheme exhibits measurable performance degradation caused by network contention, but it can be alleviated by an informed static partitioning of tasks and the addition of atomic counters to every process or compute node [7]. GTFock takes this notion one step further by following the static partitioning phase with *work stealing*. During the local phase, each process only accesses the local memory counter; however, during the work stealing phase, the process accesses other counters remotely. As illustrated in Algorithm 2, each process in GTFock begins an SCF iteration by prefetching the necessary tiles from the global density matrix and storing them into a local buffer. After all local tasks are computed, the global Fock matrix is updated. Then, each process probes the `nxtval` counters of remote processes, looking for work to steal. As we will see in Section IV-B, this algorithm results in many more *local* fetch-and-adds than *remote*, which has important performance implications for how the operations take place.

F. Making Progress in PGAS Runtimes

Modern network hardware often supports RDMA (remote direct memory access), which can transfer contiguous data without involving the remote CPU at the destination. But for more complex remote operations such as accumulates, non-contiguous data copies, and matrix multiplications, the remote CPU needs to participate in processing the tasks. In addition, the GASNet communication library used by UPC++ also requires polling the network regularly to guarantee progress. For example, in Algorithm 2, if process A is busy doing computation in line 5 while process B is trying to steal a task in line 9, then GASNet on process A may be unable to make progress until the main application thread can reach line 4 to probe the runtime. This scenario unfortunately leads to work starvation on process B.

Algorithm 2 Load balance and work stealing

```
 1: Determine total number of tasks (after screening).
 2: Statically partition tasks across process grid.
 3: Prefetch data from DAarrays.
 4: while a task remains in local queue      /* fetch_add local integer */
 5:     compute task
 6: end while
 7: update Fock matrix DArray via accumulate
 8: for Every process $p$
 9:     while a task remains in $p$'s queue   /* fetch_add remote integer */
10:         get remote blocks
11:         compute task
12:     end while
13:     update Fock matrix DArray via accumulate
14: end for
```

The key design issue here is how to make CPUs *attentive* to remote requests without wasting cycles for unnecessary polling. For instance, Global Arrays usually enables asynchronous progress with a software agent that polls continuously (interrupt-driven progress is less common, but was used on Blue Gene/P, for example). This explains why NWChem usually shows the best performance when each compute node dedicates a spare core to the constantly polling helper thread [15]. However, the optimal polling rate is generally application-dependent, so UPC++ provides two polling options: 1) explicit polling by the user application; and 2) polling at regular intervals where the user controls when to start and stop. UPC++ also allows processes on the same compute node to cross-map each other's physical memory frames into its own virtual memory address space (e.g., via POSIX or System V shared memory API), which can be used to implement a process-based polling mechanism like Casper [16].

III. Design and Implementation

This section describes the design and implementation of our Hartree-Fock in UPC++. We focus on the new DArray library, which incorporates the functionality of Global Arrays into UPC++.

A. UPC++ Hartree-Fock

We base our UPC++ implementation of HF on the GTFock code because of its exceptional load balancing capabilities and support for OpenMP multi-threading. GTFock itself contains about 5,000 lines of C code, and it uses the optimized electron repulsion integral library, OptErd [17], which contains about 70,000 lines of C and Fortran.

Because the main GTFock application is written in C, porting it to use UPC++ was mostly straightforward for *computational* components of the code. In particular, many GTFock routines solely do calculations (such as two-electron integrals) with little to no communication. An example is the local computation of the Fock matrix elements, which is the computational bottleneck of the application and makes optimized use of OpenMP clauses. In some cases, our UPC++ implementation uses C++ objects in place of GTFock's C structures, but the port primarily retains the C-style memory management and data structures.

For *communication* regions however, porting the GTFock code required substantial development and even new UPC++ features. In particular, the functionality of Global Arrays needed to be created using the UPC++ model. To aid with this, we created the DArray library, which is the subject of the next section.

B. The DArray Library

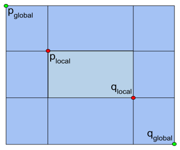
The DArray library uses UPC++ functionality to implement tiled operations on irregularly distributed arrays, like those in GTFock. This section begins by presenting how the library is used and ends with some discussion regarding the implementation. The creation of a DArray object is very similar to the creation of UPC++ multidimensional arrays:

```
/* Create a local block with rectdomain $(p_{local}, q_{local})$
in the SPMD style, then call the DArray constructor.
The rectdomain $(p_{global}, q_{global})$ defines the global
array space of dimension $M \times N$. */

/* Local array */
point<2> p_local = PT( p_row, p_col );
point<2> q_local = PT( q_row, q_col );
rectdomain<2> rd = RD( p_local, q_local );
ndarray<T, 2> local_arr( rd );
...

/* Global DArray */
point<2> p_global = PT( 0, 0 );
point<2> q_global = PT( M, N );

rectdomain<2> rd_global = RD( p_global, q_global );
DArray<T, 2> A( prows, pcols, rd_global, local_arr );
```



Now the DArray can be used for arbitrary gets and puts, even if the desired `tile` is owned by multiple ranks:

```
/* Get data residing in global rectdomain $(p, q)$,
   then place it into a local ndarray. */
ndarray<T, 2> tile;
tile = A( p, q );

/* Put local data into rectdomain rd. */
ndarray<T, 2> tile( rd );
upcxx_foreach( pt, rd ) { tile[pt] = rand(); }
A.put( tile );
```

UPC++ `async` functions also allow for custom user-defined functions to be applied to arbitrary tiles. For example, here is how we implement accumulate:

```
/* User-defined accumulate function. */
void my_accum(ndarray<T,2,global> local_block,
              ndarray<T,2,global> remote_block) {
  ndarray<T,2,global> block(remote_block.domain());
  block.copy(remote_block);
  upcxx_foreach( pt, block.domain() ) {
    local_block[pt] += block[pt];
  }; }

/* Accumulate local data into rectdomain rd. */
ndarray<T, 2> tile( rd );
upcxx_foreach( pt, rd ) { tile[pt] = rand(); }
A.user_async( tile, my_accum );
```

Finally, DArrays support simple linear algebra operations such as matrix addition, multiplication, and transpose. Currently, only two-dimensional operations are supported, because they are relevant to the HF application. However, we plan to incorporate functionality for arbitrary array dimensions in the future.

Within the DArray library itself, UPC++ multidimensional arrays are used to keep track of data ownership, make copies to/from restricted rectdomains when necessary, and manipulate local data blocks. For instance, the `DArray::transpose()` method first performs a local `ndarray::transpose()` on each block, then makes a restricted-set copy to the ranks that own the data in the transposed view.

C. Load Balancing / Work Stealing

Our implementation of Hartree-Fock in UPC++ uses the same strategy as GTFock for load balancing as described in Section II-E and outlined in Algorithm 2. However, there are many ways to implement the dynamic task counters, and we will see in Section IV-B that these alternatives have very different performance characteristics. This section outlines the different alternatives we have explored for carrying out the counter fetch-and-adds.

1) UPC++ Shared Arrays : This naive implementation simply uses a UPC++ shared array of counters and does fetch-and-adds to the elements directly. However, this is incorrect because shared arrays do not support atomic writes without locks. If multiple processes simultaneously add to a shared counter, it is possible for one of the processes to overwrite the counter with a stale value. We describe this version here only to clarify that the operation is not trivial.

2) UPC++ asyncs : This approach uses UPC++ `async` functions to increment the values of a shared array or some element of a global DArray. Because `async`'s are enqueued at the target process, they are executed atomically.

3) GASNet active messages : GASNet active messages (AM's) allow user-defined handlers to execute at a target process with the message contents passed as arguments. UPC++ `async` functions are built on top of GASNet AM's, but they carry more infrastructure to enable greater flexibility (for instance, an `async` can launch GPU kernels or contain OpenMP pragmas). On the other hand, GASNet AM handlers are more lightweight than `asyncs`, which makes them a good candidate for a simple operation like fetch-and-add. UPC++ has an experimental feature for launching such an AM with the `upcxx::fetch_add()` function.

4) GASNet GNI atomics : The Cray Gemini and Network Interface (GNI) and Distributed Shared Memory Application (DMAPP) interface provide access to one-sided communication features available to Cray Aries and Gemini network interface controllers (NIC's). For instance, 8-byte aligned atomic memory operations, such as fetch-and-add, have native hardware support on the NIC itself. They also have an associated cache for fast accesses to such data, which is particularly efficient for remote work stealing. However, accessing counters resident in a *local* NIC cache has relatively high overhead compared to accessing DRAM. GASNet does not yet expose such network atomics, but for this work we introduce a prototype implementation of fetch-and-add on 64 bit integers, anticipated to be included in GASNet-EX, the next-generation GASNet API.

Algorithm 3 Load balance and work stealing

```
Statically partition tasks and prefetch data (steps 1-3 of Alg. 2).
while a task remains in local queue
    upcxx::progress_thread_start()
    compute task
    upcxx::progress_thread_pause()
end while
update Fock matrix DArray via accumulate
for Every process $p$
    while a task remains in $p$'s queue
        get remote blocks
        upcxx::progress_thread_start()
        compute task
        upcxx::progress_thread_pause()
    end while
    update Fock matrix DArray via accumulate
end for
upcxx::progress_thread_stop()
```

D. *Attentiveness and Progress Threads*

As discussed in Section II-F, making good progress in the PGAS runtime is very important. This is particularly true while a process attempts to steal work from a victim that is busy doing computation. In order to improve the attentiveness while the application executes Algorithm 2, we added a progress thread *start/stop* feature, which the application may use to initiate runtime progress polling. In its initial implementation, the *progress_thread_start()* function spawned a pthread that calls *gasnet_AMpoll()* intermittently. We experimented across several orders of magnitude for the polling frequency and chose every 10 μs for the Hartree-Fock application. The user should call the start function just before the process will do computation and will *not* do any communication. If communication occurs before calling *progress_thread_stop()*, it requires a thread-safe version of GASNet.

Section IV-C presents measurements that suggest that calling *pthread_join()* at the end of each task incurs too much overhead for the HF application because there are potentially very many small tasks per process. We therefore introduce the *progress_thread_pause()* function. This function makes use of *pthread_cond_wait()* to sleep the progress thread and block it on a condition variable, whereas *pthread_join()* typically busy waits on a mutex (as in the GNU C Library implementation), consuming valuable CPU resources. In this new version, *progress_thread_start()* spawns the progress thread upon the first call, then signals the thread to unblock and continue polling in subsequent calls. Algorithm 3 shows the correct placement of these progress thread controller functions.

IV. Measurements and Results

This section highlights the performance measurements of our UPC++ HF application and compares it with GTFock, which uses Global Arrays and ARMCI. All experiments were run on the *Edison* supercomputer at the National Energy Research Scientific Computing Center (NERSC). Edison is a Cray XC30 petaflop system featuring the Aries interconnect with a Dragonfly topology. Edison compute nodes contain two Intel® Xeon® E5-2695 processors with two-way Hyper-Threaded cores for a total of 48 “logical cores” per compute node. Each node has 64 GB memory.

For software, our experiments were run with the default Edison module for the Intel programming environment (5.2.56 and -O3), Cray LibSci for BLAS, LAPACK, and ScaLAPACK, and a custom build of the development version of GASNet preceding the 1.26.0 release. Global Arrays is the 5.4b version linked with LibSci (and peigs disabled).

A. *Single Node Performance*

We begin with a single-node performance exploration of UPC++ and OpenMP across sensible combinations of processes and threads. The data are shown in Figure 3, where each measurement is the total time spent in an SCF iteration (averaged across 5 runs). Black boxes are not possible to run on Edison compute nodes because the job scheduler does not allow oversubscribing. The best performance is usually seen when using more threads relative to processes (except in the pathological case of only 1 process on 1 CPU socket). This is a particularly exciting result because we know that memory consumption is best with fewer UPC++ processes. Also, this property is favorable for performance on many-core architectures [18]. Other codes, such as NWChem, only exhibit good performance with 1 single-threaded process per core [15], which is a troubling characteristic in light of the prominent adoption of many-core architectures.

The best performance is seen along the diagonal, for which all executions exploit Hyper-Threading. The absolute best performance is with 2 processes (1 per socket) and 24 OpenMP threads per process. Therefore, all of our scaling experiments below are run with this configuration.

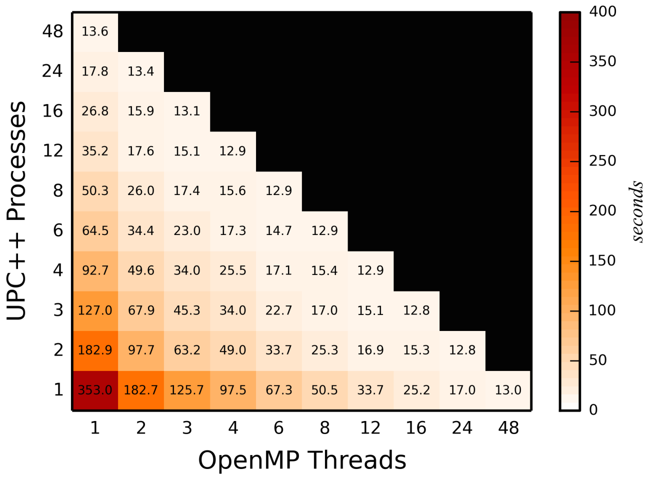


Figure 3: Single-node performance of the UPC++ HF application with UPC++ processes vs. OpenMP threads.

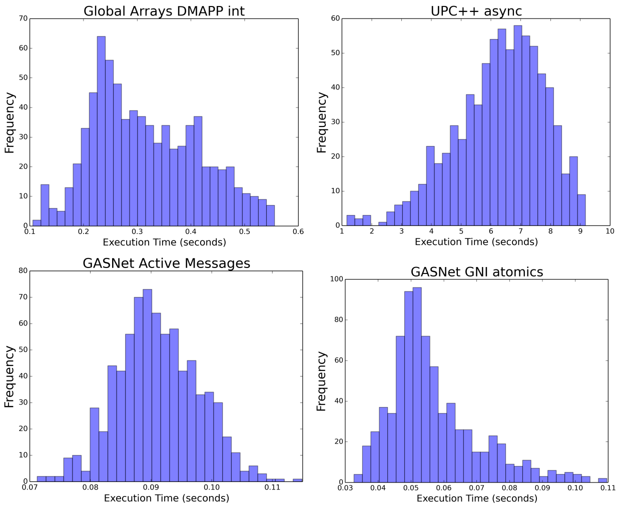


Figure 4: TAU profile data comparing the time in `nxtval` for Global Arrays (upper left), UPC++ `asyncs` (upper right), GASNet AM's (lower left), and GASNet GNI atomics (lower right). The experiment is $C_{40}H_{82}$ with 768 processes.

B. Load Balancing / Work Stealing

Our initial implementation of the task counters for load balancing/work stealing (described in Section III-C) used UPC++ `asyncs`. However, TAU profile data, in the upper-right histogram of Figure 4, shows this method incurs too much overhead for such a lightweight operation. This motivated a microbenchmark analysis for the various implementation options described in Section III-C. The microbenchmark is simple: each process owns a counter, and the benchmark ends when all counters reach 1 million. We create two different versions: one in which only *local* counters are incremented, and one in which random *remote* counters are incremented. The results are shown in Figure 5.

The most important feature of Figure 5 is that the GNI hardware-supported atomic operations on Aries show extremely good performance for *remote* fetch-and-adds, but relatively poor performance for *local* fetch-and-adds. This makes sense: GNI atomics must probe the NIC cache line for the counter, even if it is local. UPC++ `asyncs` and GASNet AM's access main memory and therefore exhibit less latency overhead. We note here that GTFock uses the `C_INT` type (4 byte integers) for task counters. However, in the ARMCI backend, this does not use the DMAPP implementation of hardware-supported atomics on Aries. It is a simple fix to use type `C_LONG`, which immediately shows much better remote fetch-and-add performance.

Due to the nature of Algorithm 2, the HF application does ∼90% local fetch-adds and ∼10% remote fetch-adds in high performing executions. Therefore, when comparing the in-application performance of GNI atomics to GASNet AM's in Figure 4, we do not see a drastic improvement. However, there is a slight benefit to using the atomics overall. Therefore, in our scaling studies in section D below, we use the GNI implementation.

C. Progress and Attentiveness

We noticed in TAU profiles (not included due to space constraints) that *pthread_join()* consumes too much execution time when calling the stop function in every iteration of the task loop. This overhead is enough to degrade performance, particularly when the task granularity is small and there are a large number of tasks. The effect is particularly noticeable when running with a large number of processes, since the relative time of the overhead eventually overshadows the time spent doing the two-electron integrals. To alleviate this effect, we added the pause feature described in Section III-D. This optimization is included in our strong scaling measurements with a polling rate of 10 microseconds. We also use a PAR build of GASNet because we require its thread-safety and do not observe any effect on the performance of the HF application.

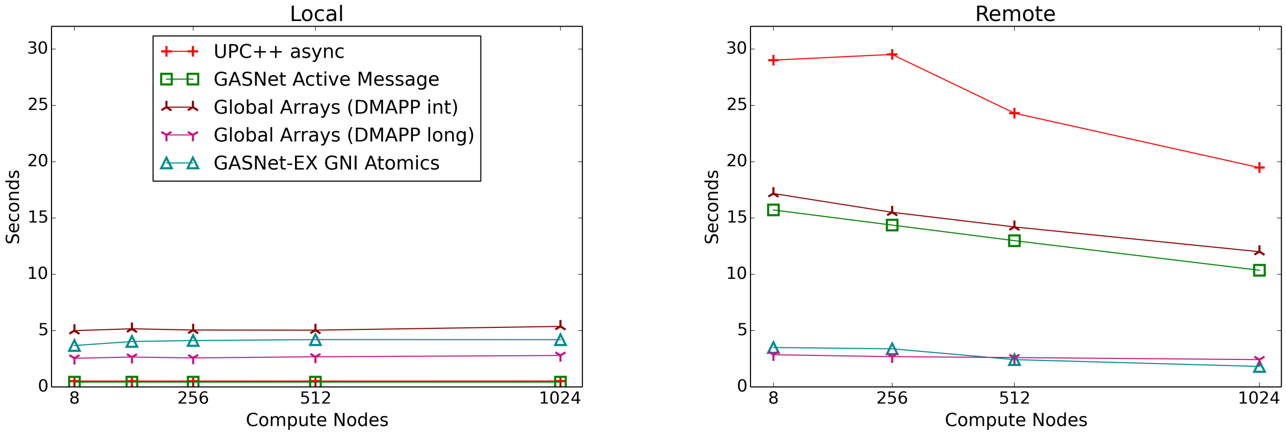


Figure 5: Flood microbenchmark with 1 million fetch/add calls per process - local (left) and remote (right) versions.

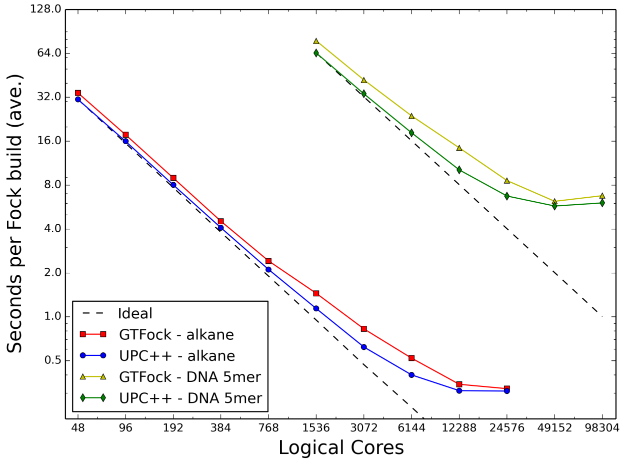


Figure 6: Strong scaling of UPC++ HF compared to GTFock and GA on Edison for two molecules: $C_{40}H_{82}$ and a DNA 5-mer. The "ideal" curves are projected with respect to the UPC++ execution at 1 node and 32 nodes, respectively.

D. Strong Scaling

With the preceding optimizations made to the UPC++ HF application, we now compare performance to the original GTFock implementation using Global Arrays and ARMCI on the Edison cluster. We look at two molecules: an alkane polymer ($C_{40}H_{82}$) and a DNA 5-mer, both using the cc-pVDZ Dunning basis set. The alkane requires 970 basis functions and the DNA 5-mer requires 3,453. Both are run with the configuration that gives the best performance: 1 process bound to each socket and 24 OpenMP threads per process. The minimum of multiple runs (2-5) is reported to reduce noise due to system variation. In summary, UPC++ HF achieves 10-20% performance improvement, with the best gains occurring in the region just before strong scaling inevitably dwindles.

V. Related Work

Related work explores relatively simple Hartree-Fock implementations in other PGAS languages like Fortress, Chapel, and X10 [19], [20]. The work from [19] presents interesting implementations in all 3 languages, but unfortunately deferred performance results to future work. The work from [20] reports performance measurements, but only for static partitioning, saying that the dynamic load balancing implementation does not scale, and using the X10 `async` for spawning tasks consistently runs out of memory.

Alternative communication infrastructures for Global Arrays have been explored in projects such as ARMCI-MPI [21]. Also, in [22] Gropp et al. present an edifying version of Global Arrays written directly with MPI-3, along with several versions of `nxtval`, including a threaded implementation.

Our UPC++ HF application has inherited the diagonalization-free purification technique for calculating the density matrix [23] from GTFock. The purification code is written in MPI, and the fact that our application uses it highlights the inter-operability of UPC++ and MPI. However, to keep things simple, our performance measurements have not included time spent in purification (except in Figure 3). The numerical effectiveness of this approach compared to classic diagonalization methods is left as future work. This is important because it affects the convergence behavior of the SCF algorithm, which will ultimately determine how well our UPC++ HF application will compare in performance to NWChem.

Related load balancing research includes resource sharing barriers in NWChem [24], inspector-executor load balancing in NWChem [7], exploiting DAG dependencies in the tensor contractions [12], and performance-model based partitioning of the fragment molecular orbital method [11]. Related work in optimizing runtime progress includes Casper, which was used to improve progress performance in NWChem [16]. It uses a process-based design that dedicates a few CPU cores to assist in communication progress of other processes, and it shows a performance benefit over traditional thread-based schemes with continuous polling. Future work might consider a comparison with our user-specified thread start/-pause/stop approach.

VI. Conclusion

Our results demonstrate that a highly tuned Hartree-Fock implementation can deliver substantial performance gains on top of prior algorithmic improvements. The performance analysis and optimization techniques presented in this paper are also applicable to a broad range of use cases that would benefit from dynamic work stealing and a global view of distributed data storage. Looking forward, we believe that both novel algorithm design and sophisticated implementation optimization are crucial to scaling real applications on upcoming parallel systems with heterogeneous many-core processors, deep memory levels, and hierarchical networks.

To facilitate tiled operations on irregularly distributed arrays, we designed and developed the DArray library*, which is capable of applying tiled gets, puts, and user-defined functions across irregularly distributed arrays containing elements of any type. DArray will be an important and reusable building block for many other similar kinds of computational problems. In future work, we plan to further enhance it by: 1) adding conditional gets on DArray blocks, where only non-zero blocks are transferred after screening; 2) optimizing other initialization-only operations (e.g., DGEMM); and 3) including other features such as diagonalization and LU decomposition. Finally, our promising OpenMP performance measurements motivate a follow-up performance analysis in preparation for the deployment of the *Cori* system at NERSC, which will equip over 9,300

Intel Knights Landing processors with the Cray Aries high-speed Dragonfly topology interconnect.

Acknowledgment

D. Ozog is supported by the Department of Energy (DOE) Computational Science Graduate Fellowship (CSGF) program. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the DOE Office of Science under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Research at the University of Oregon was supported by grants from the U.S. DOE Office of Science, under contracts DE-SC0006723, DE-SC0012381, DE-SC0005360. Finally, the authors would like to sincerely thank Dr. Edmond Chow and Dr. Xing Liu for their assistance in deploying and understanding GTFock.

References

[1] I. T. Foster, J. L. Tilson *et al.*, "Toward high-performance computational chemistry: I. Scalable Fock matrix construction algorithms," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 109–123, 1996.

[2] R. J. Harrison, M. F. Guest *et al.*, "Toward high-performance computational chemistry: II. A scalable self-consistent field program," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 124–132, 1996.

[3] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[4] S. French, Y. Zheng, B. Romanowicz, and K. Yelick, "Parallel Hessian assembly for seismic waveform inversion using global updates," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 753–762.

[5] H. Shan, B. Austin, W. De Jong, L. Oliker, N. Wright, and E. Apra, "Performance tuning of Fock matrix and two-electron integral calculations for NWChem on leading hpc platforms," in *High Performance Computing Systems. PMBS*, 2014, vol. 8551, pp. 261–280.

[6] X. Liu, A. Patel, and E. Chow, "A new scalable parallel algorithm for Fock matrix construction," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 902–914.

[7] D. Ozog, J. Hammond, J. Dinan, P. Balaji, S. Shende, and A. Malony, "Inspector-executor load balancing algorithms for block-sparse tensor contractions," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 30–39.

[8] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[9] A. Kamil, Y. Zheng, and K. Yelick, "A local-view array library for partitioned global address space C++ programs," in *ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, 2014.

[10] "GASNet home page," http://gasnet.lbl.gov.

[11] Y. Alexeev, A. Mahajan, S. Leyffer, G. Fletcher, and D. Fedorov, "Heuristic static load-balancing algorithm applied to the fragment molecular orbital method," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–13.

[12] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan, "A framework for load balancing of tensor contraction expressions via dynamic task partitioning," in *SC 2013*, Nov 2013, pp. 1–10.

[13] C. L. Janssen and I. M. Nielsen, *Parallel computing in quantum chemistry*. CRC Press, 2008.

[14] A. Vishnu, J. Daily, and B. Palmer, "Designing Scalable PGAS Communication Subsystems on Cray Gemini Interconnect," in *High Performance Computing (HiPC), 2012 19th International Conference on*, Dec 2012, pp. 1–10.

[15] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony, "Performance characterization of global address space applications: a case study with NWChem," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 2, pp. 135–154, 2012. [Online]. Available: http://dx.doi.org/10.1002/cpe.1881

[16] M. Si, A. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 665–676.

[17] N. Flocke and V. Lotrich, "Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations," *Journal of computational chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.

[18] E. Chow, X. Liu, S. Misra, M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X.-K. Liao, and P. Dubey, "Scaling up Hartree-Fock calculations on Tianhe-2," *International Journal of High Performance Computing Applications*, 2015.

[19] A. Shet, W. Elwasif, R. Harrison, and D. Bernholdt, "Programmability of the HPCS Languages: A case study with a quantum chemistry kernel," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–8.

[20] J. Milthorpe, V. Ganesh, A. Rendell, and D. Grove, "X10 as a parallel language for scientific computation: Practice and experience," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011, pp. 1080–1088.

[21] J. Dinan, P. Balaji, J. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays PGAS model using MPI one-sided communication," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 739–750.

[22] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press, 2014.

[23] A. H. R. Palser and D. E. Manolopoulos, "Canonical purification of the density matrix in electronic-structure theory," *Phys. Rev. B*, vol. 58, pp. 12 704–12 711, Nov 1998.

[24] H. Arafat, P. Sadayappan, J. Dinan, S. Krishnamoorthy, and T. Windus, "Load balancing of dynamical nucleation theory Monte Carlo simulations through resource sharing barriers," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 285–295.

[*] The UPC++, Multidimensional Array, and DArray code repositories are publicly available online at https://bitbucket.org/upcxx